AdaGIDE : A Friendly Introductory Programming Environment for a Freshman Computer Science Course

Martin C. Carlisle and A.T. Chamillard
Computer Science Department
2354 Fairchild Dr., Suite 6K41
U.S. Air Force Academy, CO 80840-6234
{mcc, achamill}@cs.usafa.af.mil

1 INTRODUCTION

We have recently transitioned the programming language in our Introduction to Computer Science course at the U.S. Air Force Academy from Pascal to Ada. Providing an intuitive and straightforward Integrated Development Environment (IDE) for Ada that is suitable for freshman use has been one of our greatest challenges. Although we recognize that a number of Ada IDEs are available, these IDEs do not seem to be designed for beginning programmers. Most of them are either too expensive for students to purchase or are designed for development of large programming projects, carrying significant overhead for the small programs we require in our freshman course. Error messages tend to be fairly complicated, assuming a relatively thorough understanding of the language syntax and semantics. Finally, both commercial and free IDEs can have extensive lead time for bug fix development and are not readily extensible.

These concerns led us to develop a Windows 95 Ada IDE that is free to the students, contains the appropriate level of functionality for our freshman course, has a minimal lead time for bug fix development (since we maintain the code in-house), and can be easily extended to contain additional help for the students. Because we implemented this environment in Ada we can also demonstrate to our students that Ada is applicable to real, large projects, and its usefulness is not limited to the small programs they create in the freshman course. This paper describes our preliminary experience with this environment.

The next section briefly describes the IDE and the third section lists a number of issues we have faced trying to interface to Windows 95. The fourth and fifth sections discuss student and faculty reaction to and use of the environment. The final section presents our conclusions and our plans for future enhancements to the IDE.

2 IDE DESCRIPTION

Our IDE, which we call AdaGIDE (for Ada Graphical IDE), uses Gnat for compiling and linking and uses a Windows 95 Graphical User Interface that is built on top of Windows libraries, particularly the Win32Ada binding. AdaGIDE includes relatively standard file manipulation (New, Open, Save, Save As, Print), editing (Undo, Cut, Copy, Paste, Find, Replace), and compilation (Compile, Build, Execute) functionality. The environment also includes several functions that do not seem to be nearly as common.

To help students recognize when they make simple typographic errors, we recently added a color-coding capability for the program text in the edit window. In other words, we use unique colors to distinguish comments, numeric constants, reserved words, and string literals from other program text (and each other).

One complaint that Pascal (and other language) programmers have about Ada is the inability to easily comment out large blocks of code, a capability that can be helpful during the debugging process. AdaGIDE provides the capability to highlight an arbitrarily large block of the program and turn the entire highlighted block into a set of comments with a single button click. Similarly, blocks of code can also be highlighted and then “uncommented”.

We also find that students tend to ignore our programming standards, which address such issues as capitalization and indentation. While it is relatively easy to penalize the students, it is still painful to grade programs that do not contain proper capitalization and indentation. In addition, we would actually prefer that our students spend their time learning and understanding Ada rather than accomplishing “busy work” to meet our programming standards. AdaGIDE therefore includes a reformat capability that can be selected by clicking a button. When reformat is selected, reserved words are changed to either upper case, lower case, or mixed case based on user preferences. The user selects from these same options for capitalization of identifiers. In addition, indentation is modified based on the lexical scope of each line in the program. We have set the defaults for reformat to our programming standards, and most students perform the reformat before submitting their programs for a grade.

3 INTERFACING TO WINDOWS 95/NT

This section discusses a number of issues we have faced interfacing to Windows 95/NT using the Win32Ada binding. Most of these issues relate to Ada's use of strong typing. Win32Ada is a thin binding to C library routines; these routines are weakly typed. As a result, our code contains many more uses of Unchecked_Access and Unchecked_Conversion than we would like. The other main difficulties encountered were the difference between C and Ada strings, and the problem of elaboration order. We begin by examining the challenges for both Windows 95 and NT, and then address NT-specific issues.

Since C procedures use pointers to local variables rather than OUT parameters, an Ada program that calls one of these procedures has two choices: either pass a pointer to a global variable, or use an Unchecked_Access to a local variable. Unchecked_Access is, in general, unsafe as you may create a pointer to a variable that will be deallocated before the pointer is; however, since we know that the C procedure will not keep a copy of this pointer, we can safely use this mechanism.

Our uses of Unchecked_Conversion mostly occur when passing a message to a Windows control. The SendMessage procedure takes 4 arguments: the handle of (pointer to) the control, an integer message identifier (e.g. GET_TEXT), and two integer parameters (wparam and lparam). Since C is not an object-oriented language, SendMessage is not overloaded for the different types of parameters the messages might require (for example, the GET_TEXT message requires a pointer to a buffer that will receive the text from the control). Instead, the pointer is cast to an integer, sent to SendMessage, and the appropriate message handler recasts the integer back to a pointer. To accomplish this in Ada, a large number of Unchecked_Conversions are used.

Another source of difficulty (and Unchecked_Conversions) is the difference between how strings are implemented in C and Ada. In C, a string is a pointer to a character; the following memory locations contain the rest of the string, up to the first zero. Ada strings have a particular size, and are not null-terminated. Fortunately, if an Ada string, S, ends with Character'First (i.e. zero), then one can simply do an Unchecked_Conversion on S(S'First)'Address to obtain a pointer to a C-style string. One must be careful doing this to avoid creating pointers to deallocated memory, as found in the following code fragment:

```
TYPE CharPointer IS ACCESS ALL Character;
FUNCTION AdaStringToC(X : IN String) RETURN Win32.LPSTR IS
  FUNCTION Convert IS NEW Ada.Unchecked_Conversion(CharPointer,
    Win32.LPSTR);
BEGIN
  RETURN Convert(X(X’First)’Address);
END AdaStringToC;

PROCEDURE HasError IS
  Y : String(1..13) := “Hello World”;
  Z : Win32.LPSTR;
BEGIN
  Z := AdaStringToC(Y & Character’First);
  -- more code that uses Z
END HasError;
```

In this code fragment, the user wishes to null-terminate the string Y, and then convert it to a C-style string so that it can be passed as an argument to a Windows procedure. Unfortunately, the memory allocated for the string `Y & Character’First` may be reused by the compiler after the call to `AdaStringToC` completes, causing the results to be unpredictable. We solve this problem by having two conversion routines. We use the above conversion routine for statically allocated strings such as globally allocated strings (unfortunately the code does not enforce this convention). We have another function that takes in an Ada string, dynamically allocates memory for the same string with the addition of a null terminator, and then returns a C-style string pointer.

The last problem we address with the Windows 95 implementation of AdaGIDE involves the order of package elaboration. Certain Windows routines (most notably `InitCommonControls`, the necessity of which we at first overlooked--causing various illegal operation errors at unpredictable locations in the program) need to be run before others. While one can use pragmas to enforce an elaboration order on the packages, we found it easier to have the main procedure explicitly initialize the controls in the appropriate order, and to use the sequence of initialization statements in the package bodies only for non-Windows related initialization.

Fortunately, once these problems were addressed, it was relatively straightforward to obtain a Windows NT executable from the Windows 95 version. The most notable difference was that

NT does not support the button style BS_BITMAP. This means that to get the pictures to appear on the buttons in the NT version, one must use an owner-drawn button type, and handle the draw item messages. Windows NT also enforces some security features for the registry and creating processes that were initially ignored in the Windows 95 version. In the end, the Windows 95 and NT executables were identical.

4 STUDENT REACTIONS

As noted above, we decided to develop AdaGIDE for a number of reasons. The environment we used last semester was designed for larger projects with multiple programmers, so there was a significant amount of overhead associated with writing even small programs. The students also felt that the error messages in this environment were vague and confusing.

Our freshman computer science course is taken by all students at the Air Force Academy, not just computer science majors. Although student response to AdaGIDE has been generally favorable, the wide diversity of student programming skills and interest has led to some surprising problems. For example, the students needed to follow a simple ten step process to install the environment, but approximately 26% were unable to accomplish the steps correctly. We plan to avoid this problem next year by pre-installing AdaGIDE on all Freshman computers, but this experience shows us that we need to continue to keep the AdaGIDE interface as simple as possible. Students also appear to expect an “industrial strength” product and act shocked when a bug is found. We develop most bug fixes in a matter of hours, but the students are hesitant to install the upgraded version from the network drive. Finally, although the Gnat error messages are more descriptive than those from the environment we used last semester, students still have difficulty interpreting them. We discuss potential solutions to this problem in the next section.

To more formally quantify student reactions to AdaGIDE, we surveyed 389 of the 453 students currently enrolled in the course (instructors did not give the survey to the remaining 64 students). A summary of that survey is provided in Table 1.

When asked how easy AdaGIDE is to use compared to other Windows 95 software (i.e., Word, Excel, etc.), 78% of the students responded that AdaGIDE is the same or easier. While we are very pleased with this result, we believe that some of the students may have interpreted

	Yes	No	Much Easier	Easier	Same	Harder	Much Harder
Were you able to successfully install AdaGIDE the first time you tried?	288 (74%)	101 (26%)					
How easy is AdaGIDE to use compared to other Windows 95 software?			5 (1%)	75 (19%)	227 (58%)	63 (16%)	19 (5%)
Do you use Reformat?	214 (55%)	175 (45%)					
If you use Reformat, is it useful?	160 (75%)	54 (25%)					
Do you use the Comment/ Uncomment buttons?	96 (25%)	293 (75%)					

Table 1. Student Survey Results

this question as "How easy is it to write a correct program compared to generating a Word document", so the ease of use may be even better than the 78% indicates.

We also asked how many students use the reformat feature. Surprisingly, 45% of the students do NOT use this feature! This result does not imply that the students manually use the correct capitalization and indentation rules, because they don't - they would apparently rather lose points on their assignments than use Reformat. To be fair, however, we must admit that there was a bug in the reformat code in the original release of AdaGIDE. This bug occasionally caused the student's program to be erased when they tried to use Reformat, so a large portion of student resistance to using Reformat could be due to this original bug (even though it was fixed several months ago). Of the 176 students who responded that they use Reformat, 25% said they don't find it useful. We infer that these 44 students use the function to avoid losing points on capitalization and indentation, but do not see any benefit (such as increased readability) from capitalization and indentation.

When questioned about use of the comment/uncomment feature, only 24% of the students responded that they use this feature. There were indications, however, that this feature was not as well-publicized as it could have been (for example, questions like "What comment/uncomment feature?" were common). This feature was included in the original release of AdaGIDE, but we did not provide user manuals, nor did we give more than a very brief tutorial on using AdaGIDE. We plan to solve this problem in the future by providing more extensive documentation.

The last question we asked in our survey was "What additional features would you like to see added to AdaGIDE?" The responses can be divided into interface functionality, code generation, and debugging help.

The first category the students would like to see added, interface functionality, includes more extensive print options (preview, page selection, 4 per page, and so on - we recently added the capability to select portions of the program for printing), right mouse click functionality (cut, paste, etc. - recently added), autosaving, automatic indenting and capitalizing (on the fly), automatic line wrapping, and spell checking. At least one student would also like a vocal natural language interface to the environment, but we've placed that additional functionality fairly low on our priority list.

The second category of additional functionality, code generation, is actually quite intriguing to several faculty members. The students have not proposed anything as extensive as true automatic code generation. Instead, several students suggested providing templates of common commands used in their code, which they could then fill in with the appropriate parameters. For example, an output statement template might look like:

```
Ada.Integer_Text_IO.Put (Item => …, Width => …);
```

where the students would replace the ellipses with the appropriate parameters. Our focus in the course is on using Ada to solve problems, so we are not particularly concerned with evaluating student memorization of language syntax. In fact, for the tests that we give on Ada, we allow students to use their textbooks to look up the syntax, so it seems reasonable to provide automated syntax help in AdaGIDE as well. We should note, however, that there is still some discussion among faculty members about whether or not providing these templates is a good idea.

The third, and by far the most common, request for additional AdaGIDE functionality was in the area of debugging help. For instance, the students would like to see more descriptive compiler error messages. We plan to provide these in a later release, but we have also noticed an interesting phenomenon in terms of student reaction to error messages. Although students request more detailed error messages, it seems as though they don't read the messages currently provided - instead, as soon as they get an error message they request help from the instructor or from a fellow student. While there are certainly some students willing to read the error messages to try to understand them, most students use the approach described above. We found this somewhat surprising, since one of our motivations for moving from our previous environment to AdaGIDE was the clarity of Gnat compiler messages. This has not provided the benefits we had expected given current student reactions to error messages. We provide an on-line help facility with descriptions, examples, and page references for Ada constructs commonly used in the course, but students have also requested more context-sensitive help, which we plan to provide (see future work). Another useful suggestion is to provide the offending line number when a program raises an exception and terminates. Unfortunately, it appears to us that this may be difficult to provide. Finally, several students requested a debugging facility in which they can step through the program, observing the values of various variables during execution. This is also planned as a future enhancement.

5 FACULTY REACTIONS

Reaction from the faculty has also been generally favorable, again with some reservations. Color coding of program text was provided in the environment we used last semester, and because many faculty members feel that including this feature enhances the usability of our IDE we added this capability in a later release. Some faculty members also feel that the first version of the environment, particularly the reformatting, should have covered the entire language and should have been bug-free. We are using an incremental approach for our development, in which we iteratively add functionality, but some faculty members would prefer to wait (forever, we suspect) for a "perfect environment". The current version of AdaGIDE contains full functionality for the freshman course, while later versions will cover more advanced features of the language.

We also surveyed 13 faculty members using the environment to more formally collect their reactions; results significantly different from student results are summarized in Table 2. Surprisingly, 23% of the faculty were unable to install AdaGIDE on the first try (compared to 26% of the students surveyed). 85% of the faculty rated AdaGIDE as the same or easier to use than other Windows 95 software, which is comparable to the 79% from the students.

	Yes	No	N/A	Much Easier	Easier	Same	Harder	Much Harder
Were you able to successfully install AdaGIDE the first time you tried?	10 (77%)	3 (23%)						
How easy is AdaGIDE to use compared to other Windows 95 software?				1 (8%)	13 (54%)	3 (23%)	2 (15%)	0
Do you use Reformat?	7 (54%)	6 (46%)						
If you use Reformat, is it useful?	7 (100%)	0						
Do you use the Comment/ Uncomment buttons?	8 (62%)	5 (38%)						
Do you like color-coding of reserved words, comments, and literals?	13 (100%)	0						
Do you prefer AdaGIDE to previous environments for this course?	13 (77%)	2 (15%)	1 (8%)					
Do you prefer AdaGIDE to other environments for upper division courses?	8 (62%)	2 (15%)	3 (23%)					

Table 2. Faculty Survey Results

For the reformat function, 46% of the faculty don't use the provided functionality (compared to 45% for students). We suspect that some instructors simply prefer to type the code in correctly as they go, but others have been frustrated by the fact that our initial reformat capability only covered the basic structures used in the course. Of the 7 faculty who do use reformat, all find it useful. 62% of the faculty use the comment/uncomment feature (compared to 25% of the students) - we suspect this is because of better communication among the faculty about this feature, and also because the faculty realize the benefit of commenting out selected portions of code during the debugging process. All of the faculty surveyed liked color-coded reserved words, identifiers, and literals.

When asked if they prefer AdaGIDE to previous environments used in the course, 10 said they did, 2 said they did not, and 1 declined to comment (this is his first semester on the faculty). For upper division courses (e.g., those for our sophomore, junior, and senior majors), 8 faculty members said they prefer AdaGIDE to other environments, 2 said they did not, and 3 did not comment. Finally, when asked to comment on response time for bug fixes, 1 faculty member was neutral (no comment), 3 rated response time as good, and 9 rated response time as very good (the highest rating). Since all bug fixes have been provided within 24 hours, we believe the 3 good ratings reflect faculty impatience with enhancements to the reformat capability.

The list of desired enhancements provided by the faculty closely matches the list provided by the students. The only additional request was for a Multiple Document Interface (MDI) for AdaGIDE. Although multiple instantiations of AdaGIDE can be run concurrently, with full cut/copy/paste functionality between them, the purists argue (correctly) that multiple instantiations would not be required if an MDI were provided. We agree, but have set this as a low priority compared to our other future work.

6 CONCLUSIONS AND FUTURE WORK

We are pleased with the progress we have made developing an Ada IDE that is suitable for use in our freshman course - in fact, we have also started using AdaGIDE in our sophomore-level programming languages course. Interfacing with Windows 95 has led to unattractive code and subtle bugs in some cases, but student and faculty reaction to the environment is generally positive. We therefore plan to continue development of the IDE by implementing additional functionality in the interface.

One such enhancement is driven by the fact that our freshman students seem to have a difficult time discerning the meaning of the Gnat error messages. While these error messages seem descriptive to the computer science faculty, it is clear that the students could use additional help interpreting the messages. Therefore, one of our future enhancements will be to provide on-line error message elaboration. By double-clicking an error message, students will be able to open a dialog box that explains potential causes of that error in further detail. Within this dialog box, we also plan to provide references to areas in the course textbook that address the syntactic construct causing the error. We believe the combination of error message elaboration and references to the course text will greatly improve the students' ability to correct syntax errors in their code.

Although many of our students believe that successful compilation is equivalent to correct program behavior, it seems that a debugging facility should also be provided in the IDE. Our future work therefore includes developing an IDE interface to an existing debugger.

Finally, we plan to expand our reformatting tool to cover more features of the Ada language (exception handling, for example).